# Solución Examen de Programación 3 y III (19/07/2014)

Instituto de Computación, Facultad de Ingeniería, UdelaR

## Parte Teórica Obligatoria

### Ejercicio 1 (10 puntos)

a)

$O(f) = \{ g: N \rightarrow R^* / \exists c \in R^+, \exists n_0 \in N, \forall n \in N, n > n_0, g(n) \leq c*f(n)\}$

$\Theta(f) = O(f) \cap \Omega(f)$

donde:

$\Omega(f) = \{ g: N \rightarrow R^* / \exists c \in R^+, \exists n_0 \in N, \forall n \in N, n > n_0, g(n) \geq c*f(n)\}$

b) a demostrar: Si $f \in \Theta(g) \Rightarrow g \in \Theta(f)$

Como $f \in \Theta(g) \Rightarrow f \in O(g)$ y $f \in \Omega(g)$ por definición de $\Theta$

Usando y demostrando la propiedad: $f \in O(g) \Leftrightarrow g \in \Omega(f)$ quedaría demostrado.

Utilizando las definiciones de O y Ω:

i) $f(n) \in O(g(n))$ si y solo si $\exists c_1 \in R^+, n_1 \in N / \forall n > n_1 \; f(n) \leq c_1.g(n)$

ii) $g(n) \in \Omega(f(n))$ si y solo si $\exists c_2 \in R^+, n_2 \in N / \forall n > n_2 \; g(n) \geq c_2.f(n)$

**(→)**

A partir de i) se tiene que: $\forall n > n_1 \; \frac{1}{c_1} * f(n) \leq g(n)$

Si se elige $n_2 = n_1$ y $c_2 = 1/c_1$:

$g(n) \geq c_2.f(n) \; \forall n > n_2$ entonces $g(n) \in \Omega(f(n))$

La demostración en el otro sentido es análoga.

### Ejercicio 2 (10 puntos)

a)

```
  void merge_sort_aux(int* v, int start, int end){
    if (start != end){
     int m = (start + end) / 2;
     merge_sort_aux(v, start, m);
     merge_sort_aux(v, m + 1, end);
     merge(v, start, m, m+1, end);
    }
  }

void merge_sort(int* v, int n){

  merge_sort_aux(v,0, n-1);

}
```

b) La estrategia es **Divide & Conquer**.

La forma en la que se aplica es la siguiente:

**Descomposición**: se divide el vector a ordenar a la mitad

**Combinación**: se combinan las soluciones con la merge.

## Ejercicio 3 (10 puntos)

Sea maxCompras(c, i, p), con :

- c, capacidad del contenedor,
- i, indice de compra a considerar
- p, monto de valor de compras disponible

  - Puedo incluir la compra i opcionalmente:
    si p >= compras[i].valor y compras[i].tamaño <= c, i en [0,N-1].
    = max(maxCompras(c,i-1,p), maxCompras(c-compras[i].tamaño, i-1,p-compras[i].precio) +1) ,

  - No puedo incluir la compra i:
    si p < compras[i].valor o compras[i].tamaño > c, i en [0,N-1]
    = maxCompras(c,i-1,p)

  - Base: No puedo incluir ninguna compra más
    si i<0 o p=0 o c=0
    = 0

## Ejercicio 4 (10 puntos)

a) Siguiendo greedy para maximizar la cantidad de elementos, se define como criterio óptimo local, la selección de la compra que ocupe menos espacio dentro de las factibles (precio y capacidad aceptables) ya que localmente es la que me dejará mayor espacio para seguir asignando compras.

```
monto = M
capacidad = C
seleccionados =[]
Mientras (lugarOcupado<capacidad && hayElementosParaElegir(compras){
 e = obtenerCompraMenorTamano (compras);
 Si (monto >= e.valor) {
        agregarASolucion(e);
        monto= monto-e.valor;
        capacidad = capacidad – e.capacidad;
        agregarSeleccionado(e)
}
}
return seleccionados;
```

b) No, el criterio de selección basado en el tamaño de la compra no garantiza el óptimo global ya que el precio incide como restricción y no es considerada. Como contraejemplo bastaría considerar un conjunto de compras donde las compras de menor capacidad tienen los precios mucho mayores y restringen la cantidad que puedo poner.

# Problemas

## Problema 1 (30 puntos)

**Forma de la solución:**

Tupla de largo variable N, T=<$t_1$, …, $t_n$>, donde N representa la cantidad de partidos a la que asistirán.

La cantidad de partidos que pueden asistir como máximo es 14 (6 de octavos + 4 de cuartos + 2 semi-finales + partido de tercer puesto + final). Esto hace que N sea menor o igual a 14.

Cada $t_i$ es un entero que representa el identificador del partido al que los amigos asistirán en el partido i.

**Restricciones explícitas:**

Cada elemento de la tupla T representa un partido NO disputado del mundial.

$$\forall i \in \{1, \dots, N\}: t_i \in \{1, \dots, 15\} \ \wedge \ t_i \neq pUruCol$$

**Restricciones implícitas:**

Para cualquier par de partidos a los que el grupo de amigos asistirá consecutivamente existe algún medio de transporte tal que partiendo desde el final del primer partido se puede llegar antes del comienzo del segundo.

$$\begin{aligned} \forall i \in \{1, \dots, N-1\} : & \ DiferenciaPartidos[t_i][t_{i+1}] \\ & > DuracionOmnibus[partidos[t_i].sede][partidos[t_{i+1}].sede] \\ \vee \ & DiferenciaPartidos[t_i][t_{i+1}] \\ & > DuracionAvion[partidos[t_i].sede][partidos[t_{i+1}].sede] \end{aligned}$$

La suma del precio de todas las entradas de los partidos y los viajes entre sedes no puede ser mayor al dinero que poseen.

$$\begin{aligned} D > \ & CostoTransporte(partidos[pUruCol].sede, partidos[t_1].sede) \\ & + \sum_{i=1}^{N-1} ( \ CostoTransporte(partidos[t_i].sede, partidos[t_{i+1}].sede) \\ & + partidos[i].entrada \ ) + partidos[N].entrada \end{aligned}$$

$$CostoTransporte(s_A, s_B) = \begin{cases} CostoAvion[s_A][s_B] & si \ \ DiferenciaPartidos[s_A][s_B] < DuracionOmnibus[s_A][s_B] \\ CostoOmnibus[s_A][s_B] & si \ \ DiferenciaPartidos[s_A][s_B] < DuracionAvion[s_A][s_B] \\ \min\{CostoAvion[s_A][s_B], CostoOmnibus[s_A][s_B]\} & en \ caso \ contrario \end{cases}$$

**Función objetivo:**

Se busca maximizar las preferencias futboleras del grupo de amigos a través de la maximización de la satisfacción según los partidos asistidos.

$$max_{T \in Soluciones} PreferenciaPartidosAsistidos(T)$$

$$PreferenciaPartidosAsistidos(T) = \sum_{i=1}^{N} partidos[t_i].preferencia$$

**Predicados de poda:**

Se realiza una poda si la suma de las preferencias de la solución parcial con las preferencias de los partidos que se disputarán posteriormente al último partido de la tupla parcial no supera la suma de preferencias de los partidos seleccionados en la mejor solución encontrada hasta el momento.

$$Siendo\ S = < s_1, \dots, s_N > la\ mejor\ solución\ encontrada\ hasta\ el\ momento\ y\ T = < t_1, \dots, t_j, \dots > \ una\ solución\ parcial$$

Entonces la solución parcial se poda si:

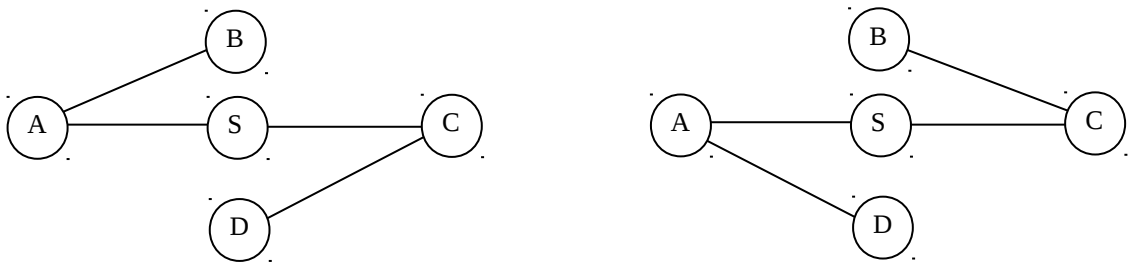
$$PreferenciaPartidosAsistidos(S) \geq \sum_{i=1}^{j} partidos[t_i].preferencia + \sum_{p \in PartidosNoDisputados} partidos[p].preferencia$$

$$siendo\ PartidosNoDisputados = \{p \in \{1,..,15\} : DiferenciaPartidos[t_j][p] > 0\}$$

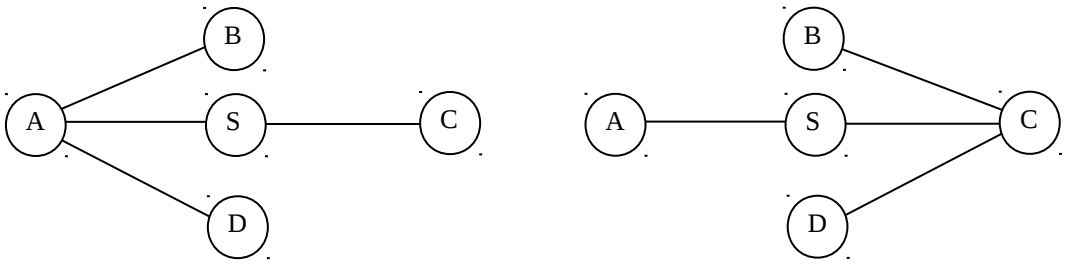
# Problema 2 (30 puntos)

a)

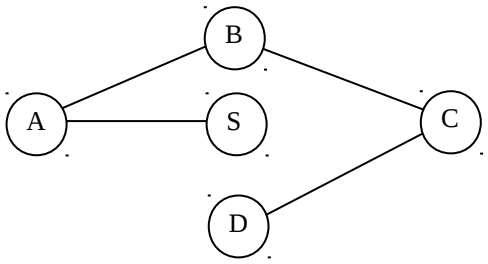
Las soluciones son:




Los siguientes son los árboles que pueden obtenerse como resultado de BFS (G,S). La longitud del camino desde S hasta cada vértice es iguales a la distancia en G entre S y el vértice.




El siguiente es un ejemplo de árbol en el que la longitud del camino de S a algunos vértices es mayor a la distancia en G.




**b)**

Se recorre g con una BFS que comienza en u, manteniendo la distancia de u a cada vértice. Si al procesar la arista (v,w) el vértice w todavía no había sido descubierto, se establece distancia (u,w) = 1 + distancia (u,v) y se inicializa cantidades (u,w) = cantidades (u,v). Si w ya había sido descubierto y distancia (u,w) = 1 + distancia (u,v), entonces los caminos de longitud mínima de u a v, concatenados con (v,w) también son caminos de longitud mínima de u a w. Entonces, a cantidades (u,w) se suma cantidades (u,v). En otro caso, los caminos que llevan a w a través de v no son de longitud mínima y no se hace nada.

```
int * caminosMinimos (Grafo g, int n, int u)
{
      bool * distancias = new int [n + 1]; // distancias a u
      bool * cantidades = new int [n + 1]; // cantidad de caminos más cortos
      bool * descubiertos = new bool [n + 1];
      int i;
      for (i = 1; i <= n; i++)
      {
            descubiertos [i] = false;
            distancias [i] = INT_MAX;
            cantidades [i] = 0;
      }
      distancias [u] = 0; // la distancia de u a u es 0
      cantidades [u] = 1; // por definición
      Lista adyacentes;
      int v, w;
      Cola cola = CrearCola();

      Encolar (cola, u);
      descubiertos [u] = true;

      do
      {
            v = Desencolar (cola);
            adyacentes = Adyacentes (g, v);
            while (! EsVaciaLista (adyacentes))
            {
                  w = Primero (adyacentes);
                  adyacentes = Resto (adyacentes);
                  if (!descubiertos [w])
                  {
                        Encolar (cola, w);
                        descubiertos [w] = true;
                        distancias [w] = 1 + distancias [v];

                       /* cada uno de los caminos más cortos desde u a v,
                          concatenados con (v,w) son caminos más cortos desde u
                          hasta w */

                        cantidades [w] = cantidades [v];
                  }
                  else if (dists [w] = 1 + distancias [v])
                  {
```

```
                        /* los caminos que concatenan (v,w) a los caminos más
                           cortos desde u a v tienen la misma longitud que los
                           caminos más cortos hallados anteriormente */

                         cantidades [w] += cantidades [v];
                    }
                    /* en otro caso la distancia es mayor al camino más corto */
               }
          } while (! EsVaciaCola (cola));

          delete descubiertos;
          delete distancias;
          destruirCola (cola);
          return cantidades;
}
```